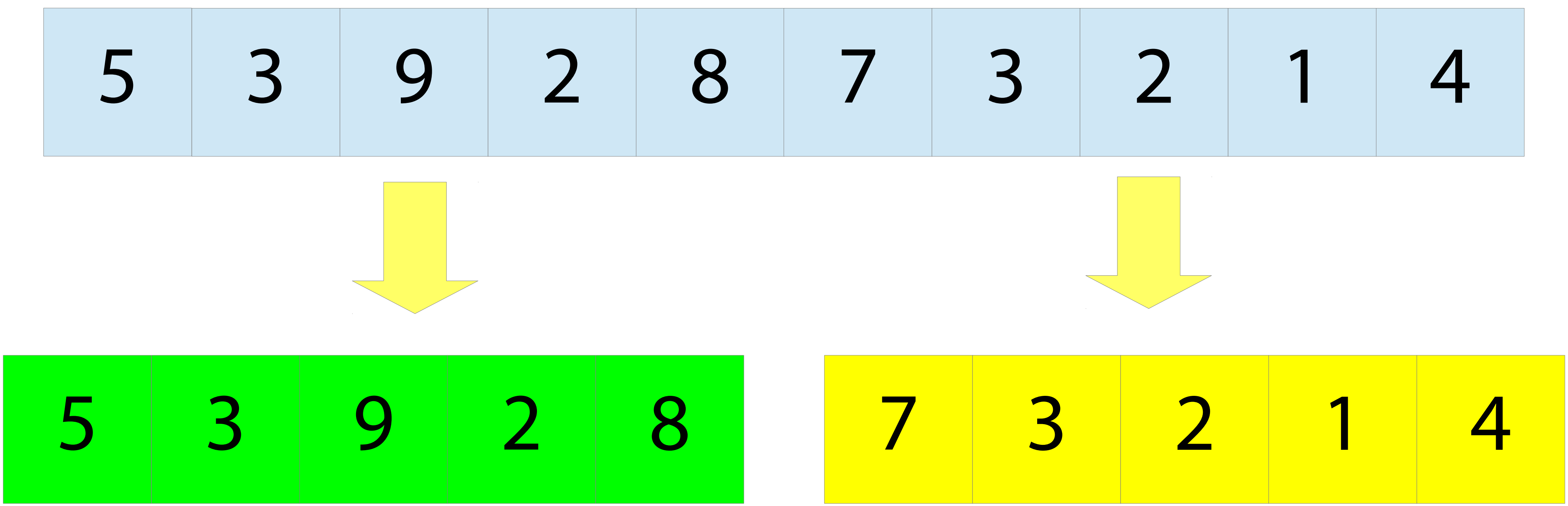
Quicksort

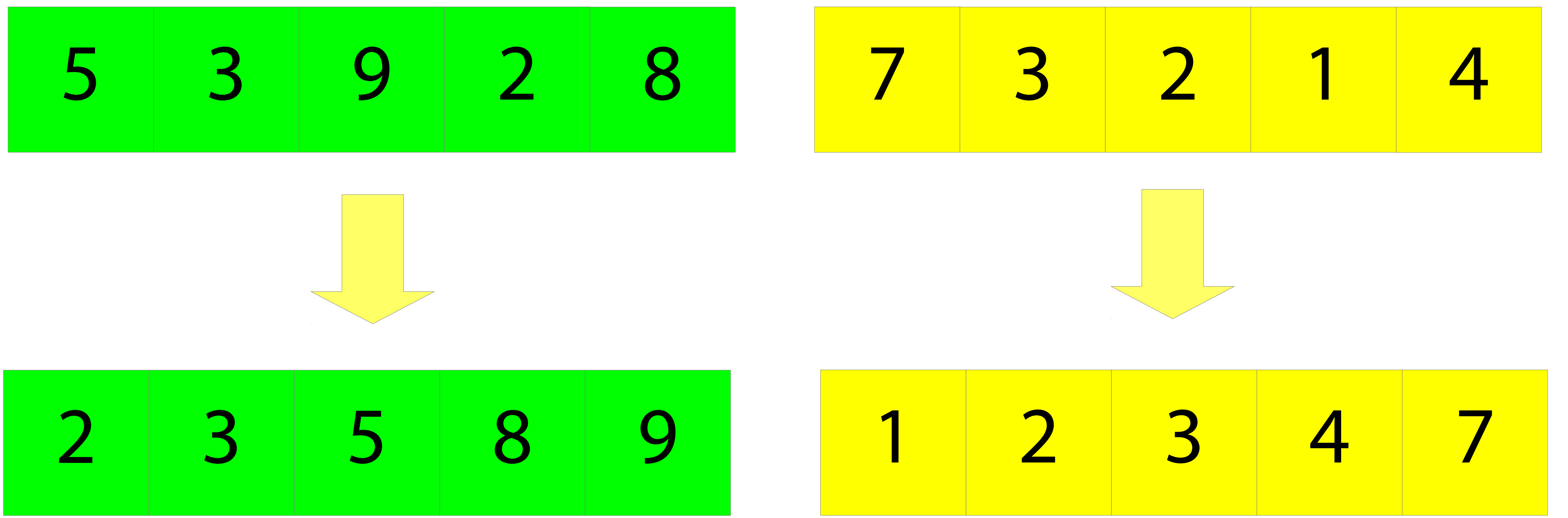
Mergesort again

1. *Split* the list into two equal parts



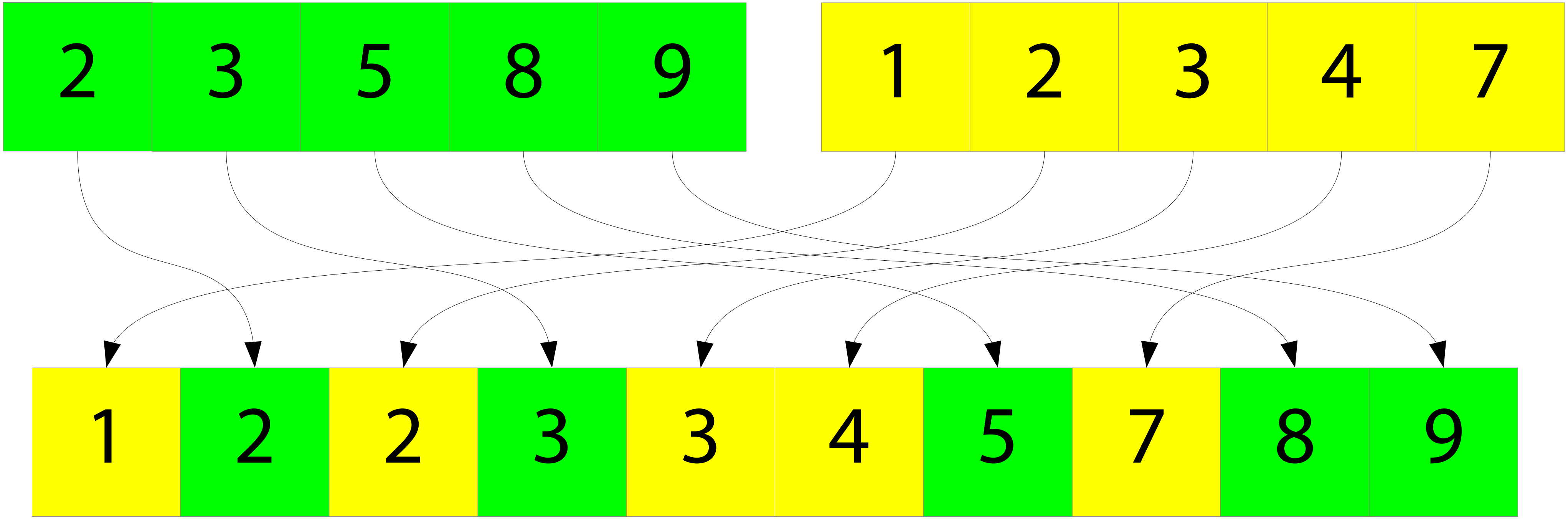
Mergesort again

2. *Recursively* mergesort the two parts



Mergesort again

3. *Merge* the two sorted lists together



Quicksort

Mergesort is great... except that it's not in-place

- So it needs to allocate memory
- And it has a high constant factor

Quicksort: let's do divide-and-conquer sorting, but do it in-place

Quicksort

Pick an element from the array, called the *pivot*

Partition the array:

- First come all the elements smaller than the pivot, then the pivot, then all the elements greater than the pivot

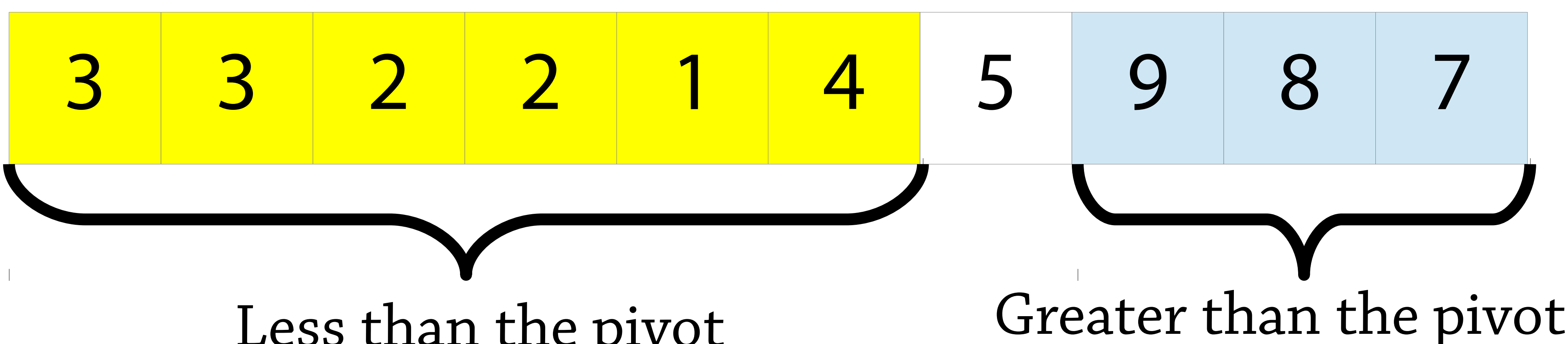
Recursively quicksort the two partitions

Quicksort

5	3	9	2	8	7	3	2	1	4

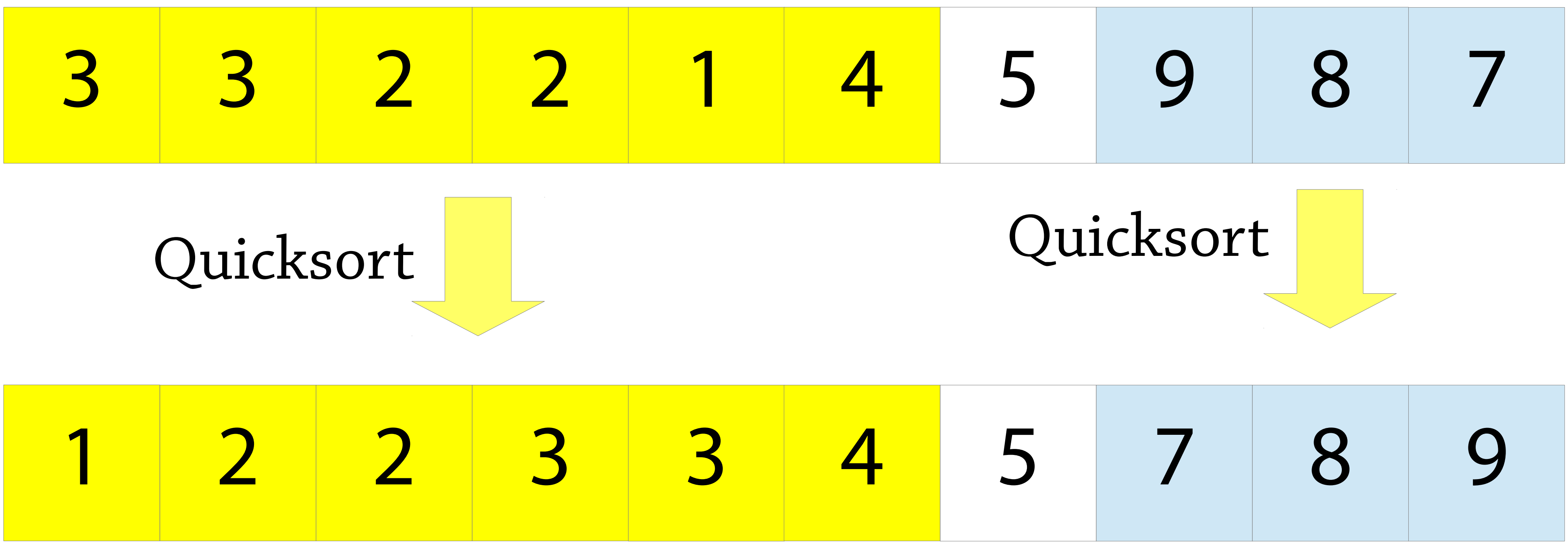
Say the pivot is 5.

Partition the array into: all elements less than 5, then 5, then all elements greater than 5



Quicksort

Now recursively quicksort the two partitions!



Pseudocode

```
// call as sort(a, 0, a.length-1);
void sort(int[] a, int low, int high) {
   if (low >= high) return;
   int pivot = partition(a, low, high);
      // assume that partition returns the
      // index where the pivot now is
   sort(a, low, pivot-1);
   sort(a, pivot+1, high);
}
```

Common optimisation: switch to insertion sort when the input array is small

Quicksort's performance

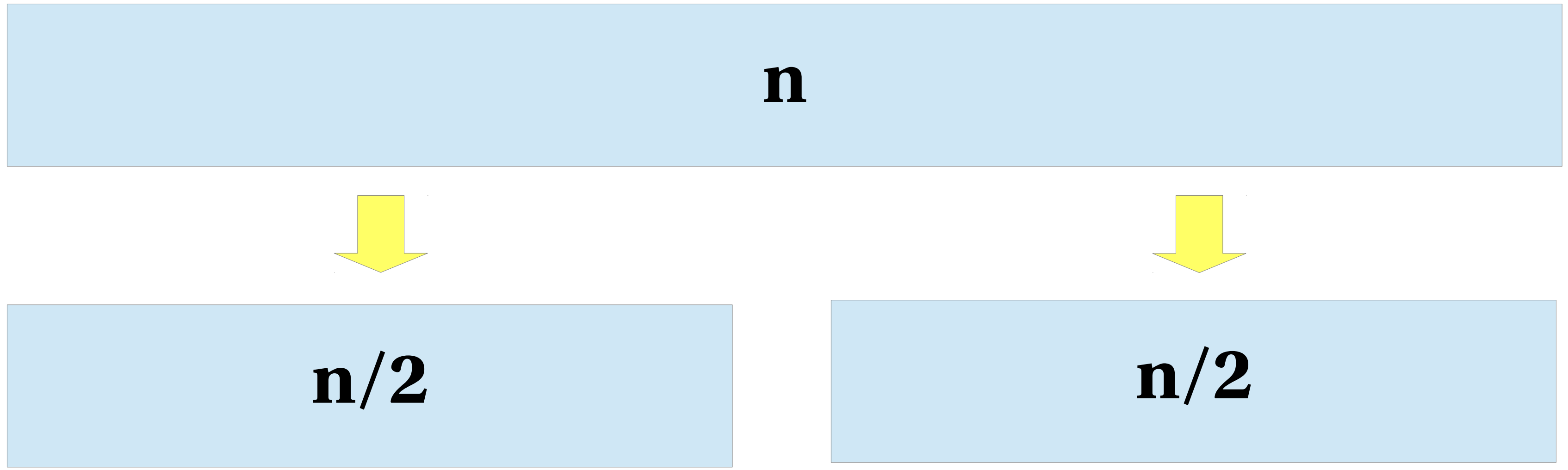
Mergesort is fast because it splits the array into two *equal* halves

Quicksort just gives you two halves of whatever size!

So does it still work fast?

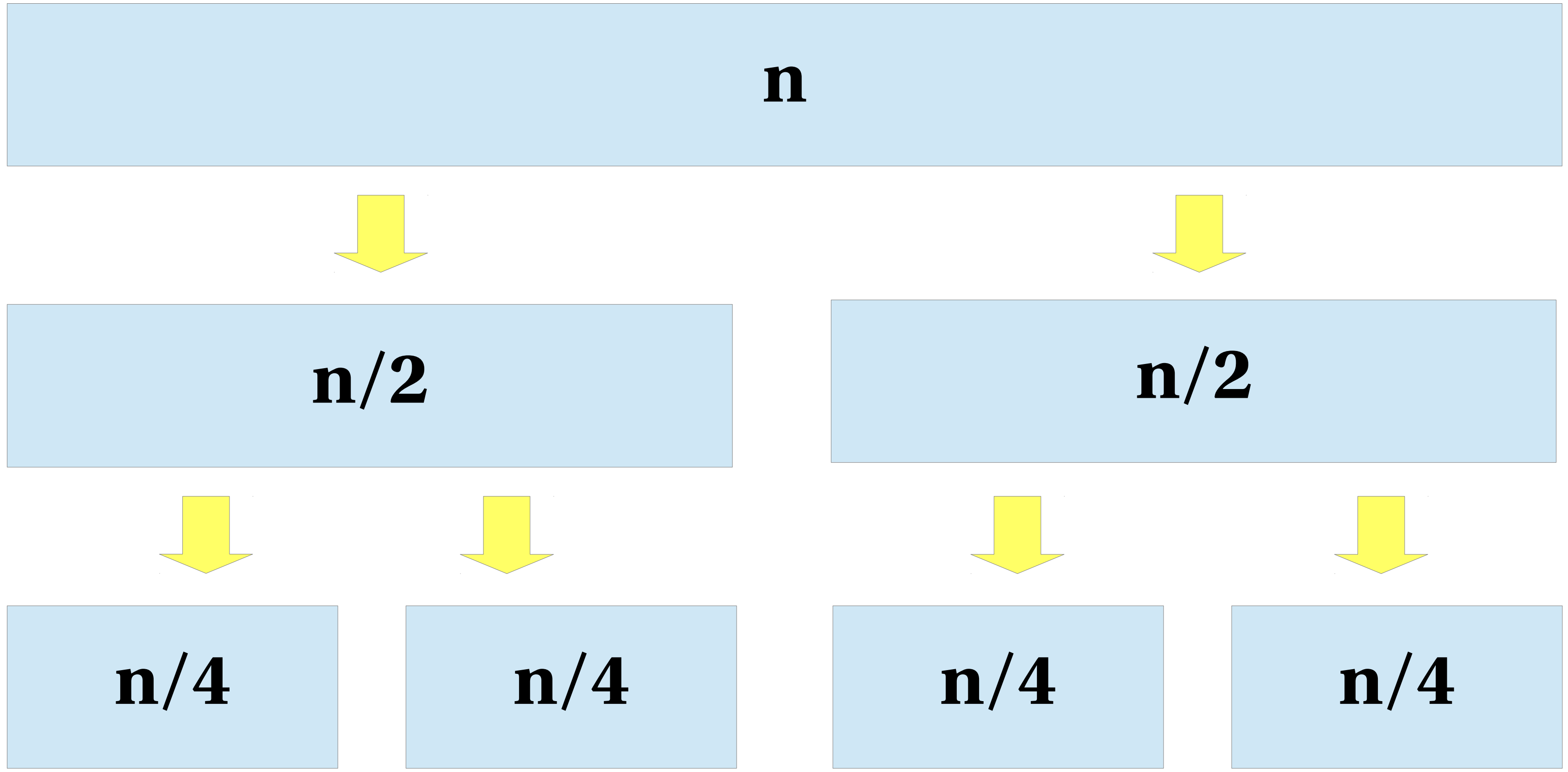
Complexity of quicksort

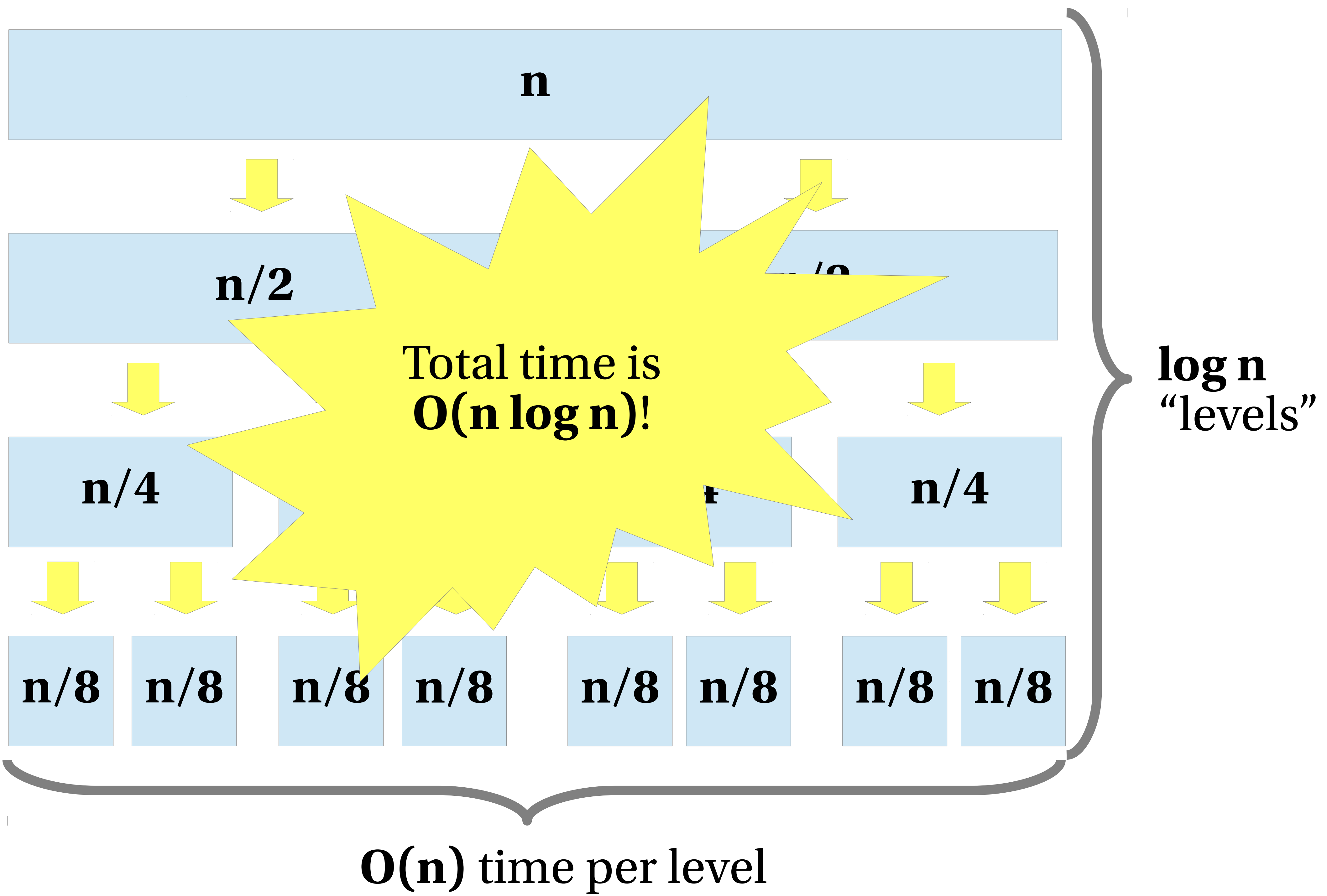
In the best case, partitioning splits an array of size n into two halves of size n/2:



Complexity of quicksort

The recursive calls will split these arrays into four arrays of size n/4:





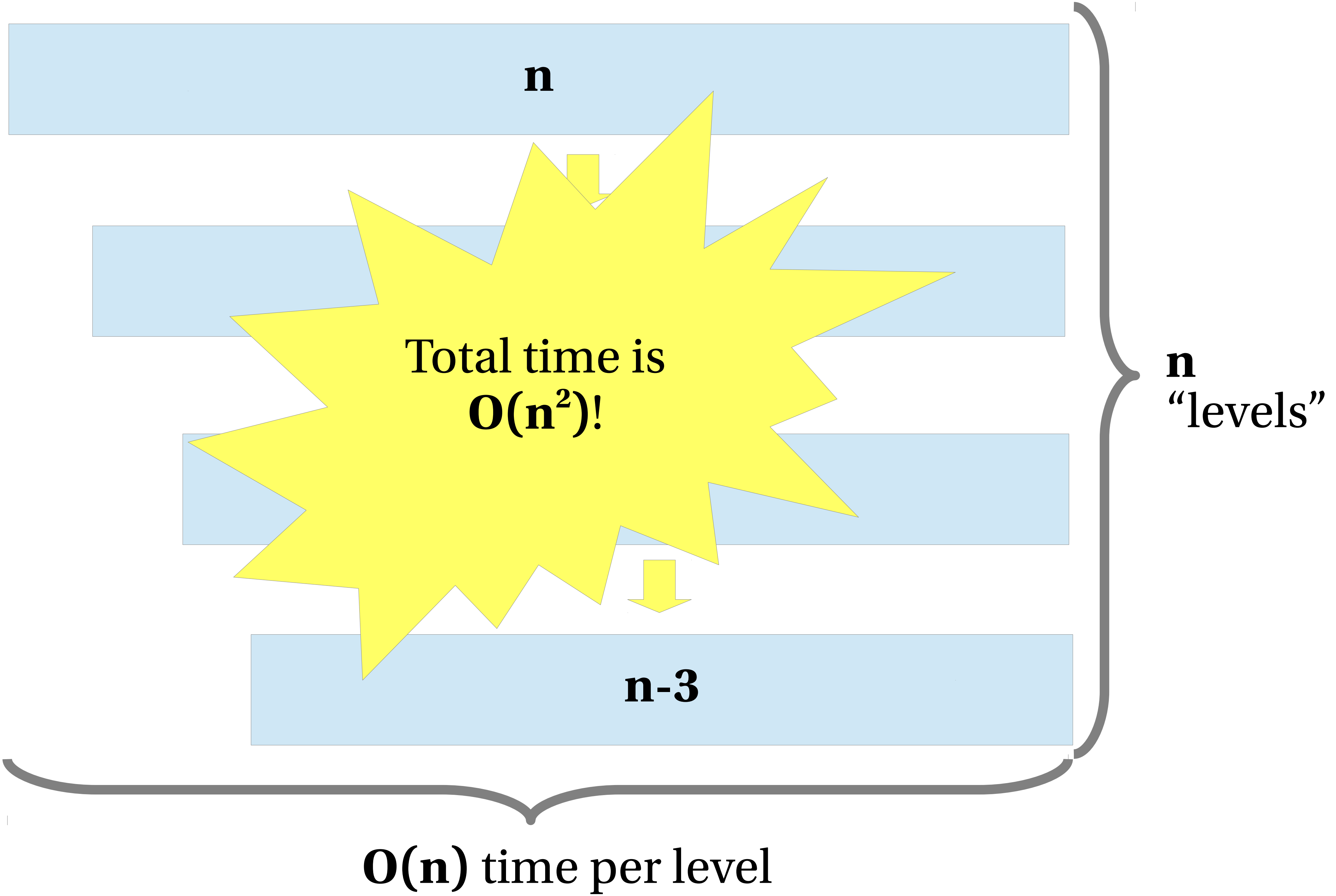
n
n/2
n/4
n/4
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
Total time is
O(n log n)!
log n
“levels”
O(n) time per level

Complexity of quicksort

But that's the best case!

In the worst case, everything is greater than the pivot (say)

- The recursive call has size n-1
- Which in turn recurses with size n-2, etc.
- Amount of time spent in partitioning:
  n + (n-1) + (n-2) + … + 1 = **$O(n^2)$**


n
Total time is
O(n²)!
n-3
n
"levels"
O(n) time per level

Worst cases

When we simply use the first element as the pivot, we get this worst case for:

- Sorted arrays
- Reverse-sorted arrays

The best pivot to use is the *median* value of the array, but in practice it's too expensive to compute…

Most important decision in QuickSort: **what to use as the pivot**

Complexity of quicksort

You don't need to split the array into *exactly* equal parts, it's enough to have some balance

- e.g. 10%/90% split still gives O(n log n) runtime
- Median-of-three: pick first, middle and last element of the array and pick the median of those three – gives O(n log n) in practice
- Pick pivot at random: gives O(n log n) *expected* (probabilistic) complexity

Introsort: detect when we get into the $O(n^2)$ case and switch to a different algorithm (e.g. heapsort)

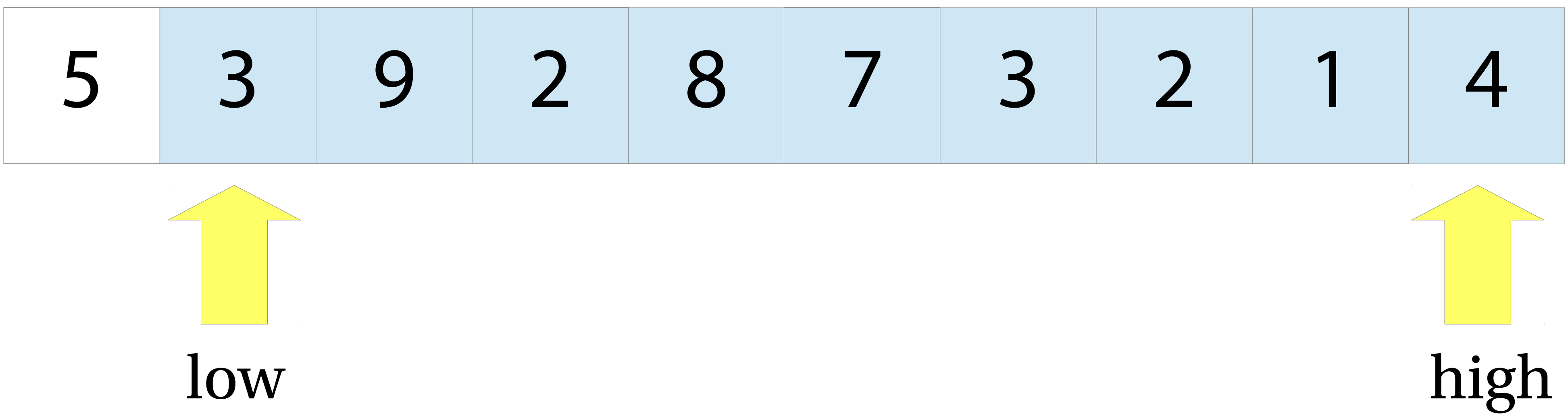
Partitioning algorithm

1. Pick a pivot (here 5)

5	3	9	2	8	7	3	2	1	4

Partitioning algorithm

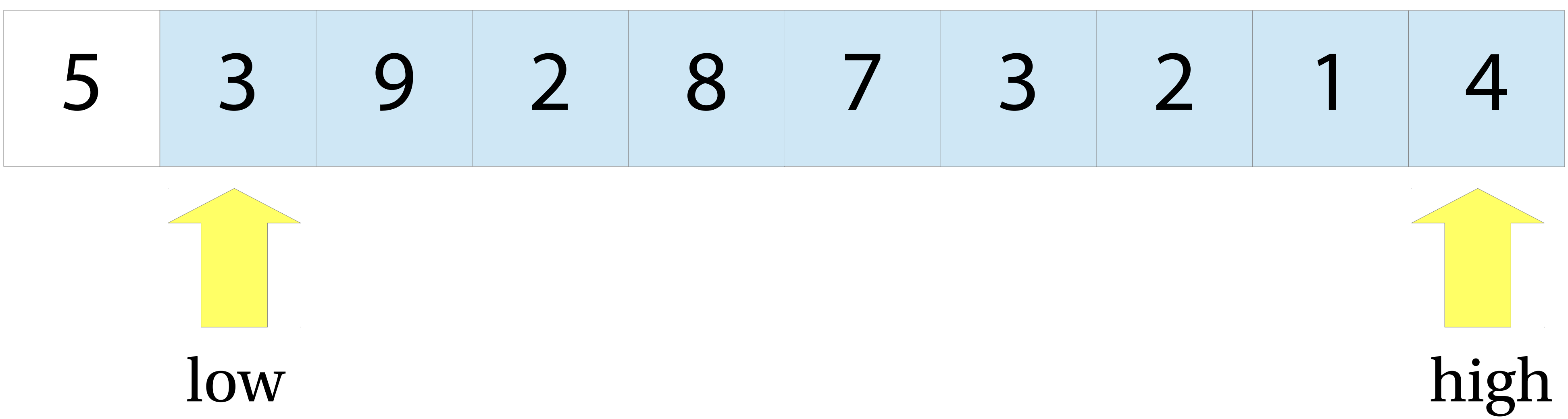
2. Set two indexes, low and high



Idea: everything to the left of low is less than the pivot (coloured yellow), everything to the right of high is greater than the pivot (green)

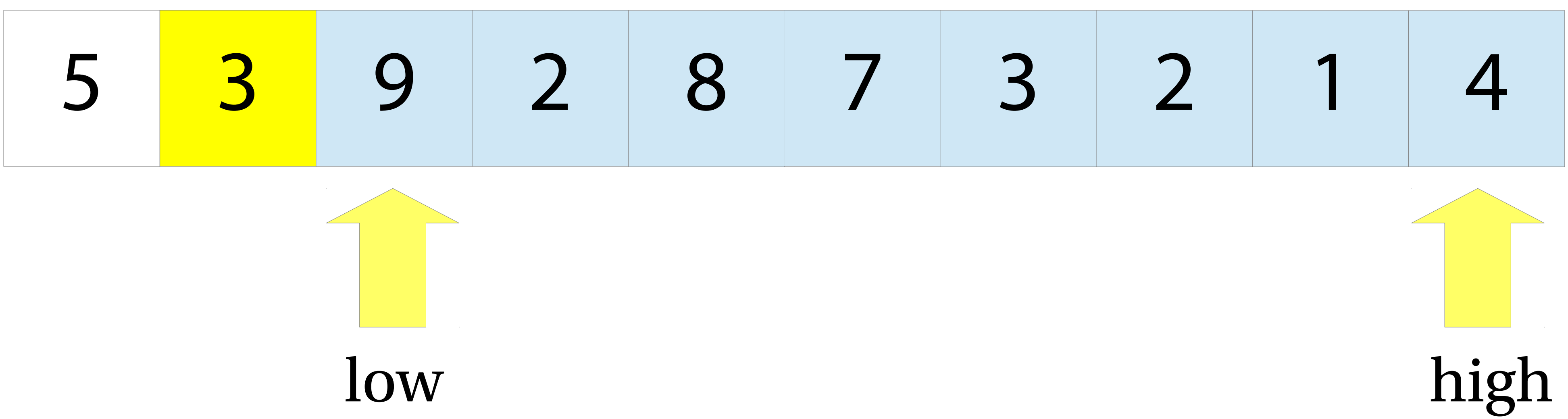
Partitioning algorithm

3. Move low right until you find something greater than the pivot



Partitioning algorithm

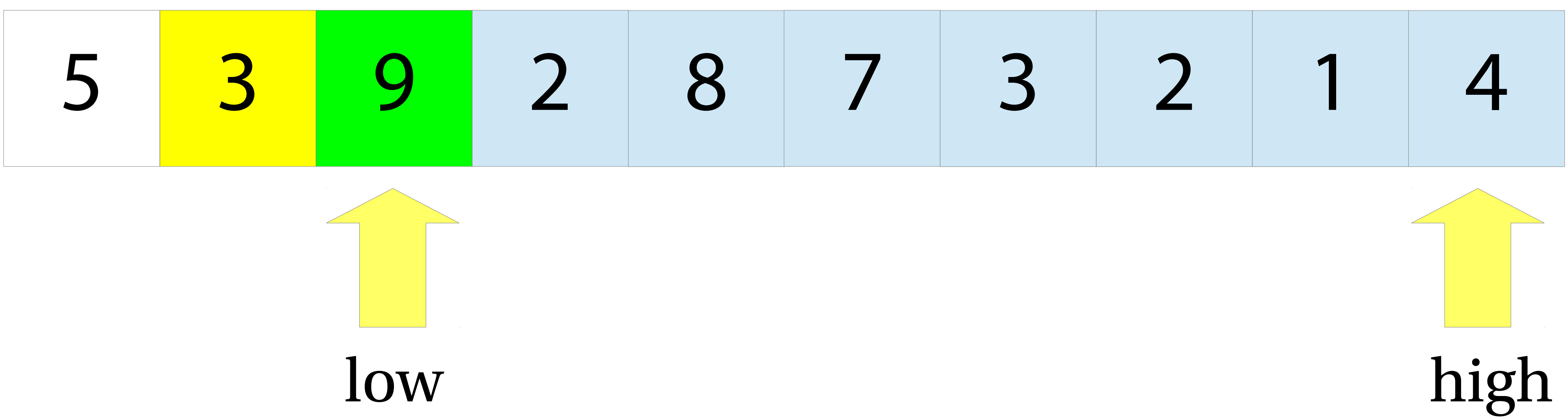
3. Move low right until you find something greater or equal to the pivot




```
while (a[low] < pivot) low++;
```

Partitioning algorithm

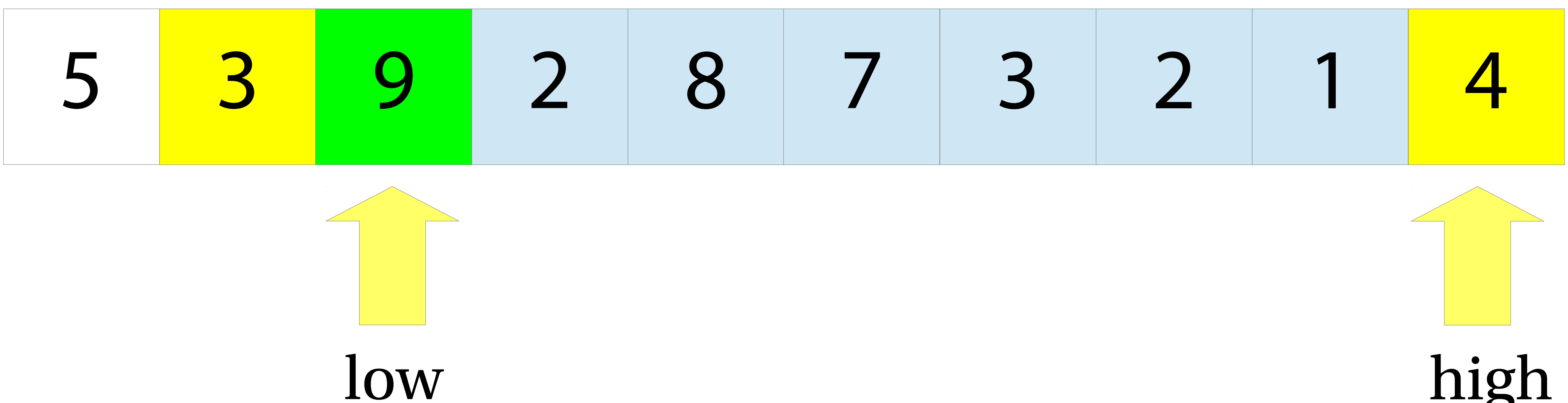
3. Move low right until you find something greater than the pivot




```
while (a[low] < pivot) low++;
```

Partitioning algorithm

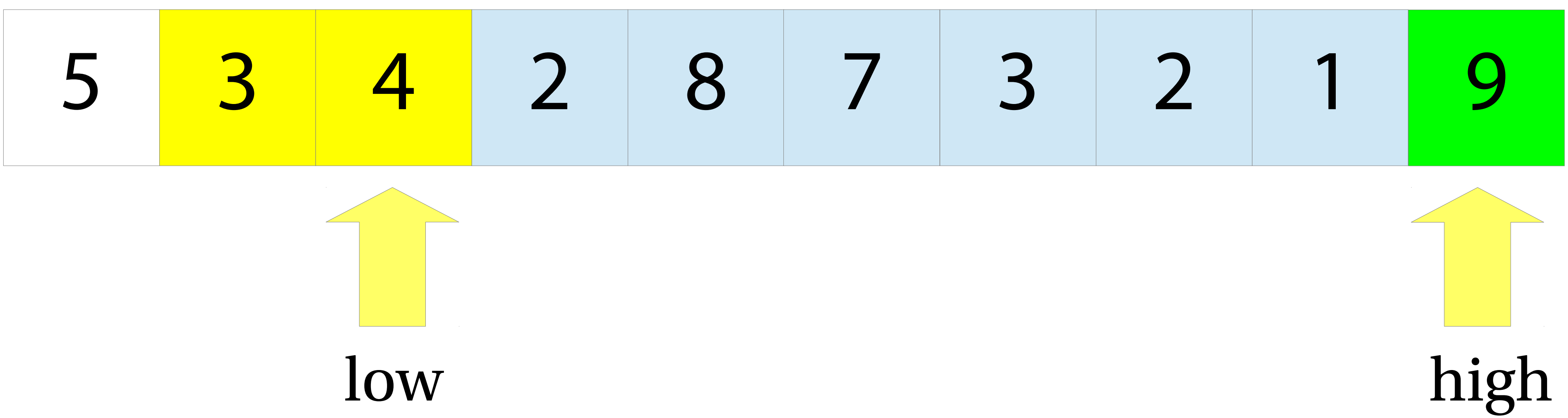
3. Move high left until you find something less than the pivot




```
while (a[high] < pivot) high--;
```

Partitioning algorithm

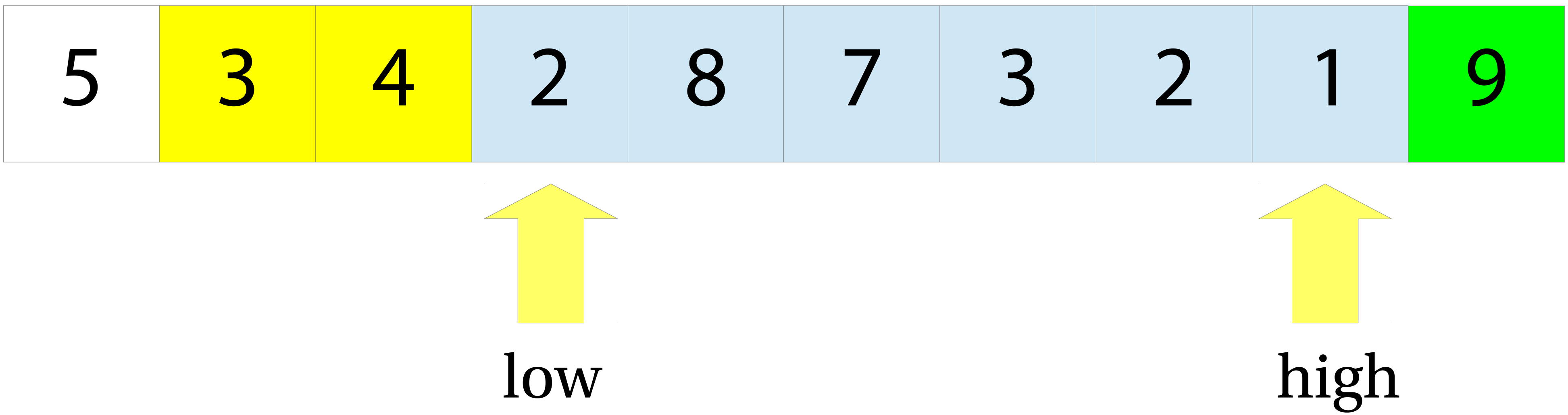
4. Swap them!




```
swap(a[low], a[high]);
```

Partitioning algorithm

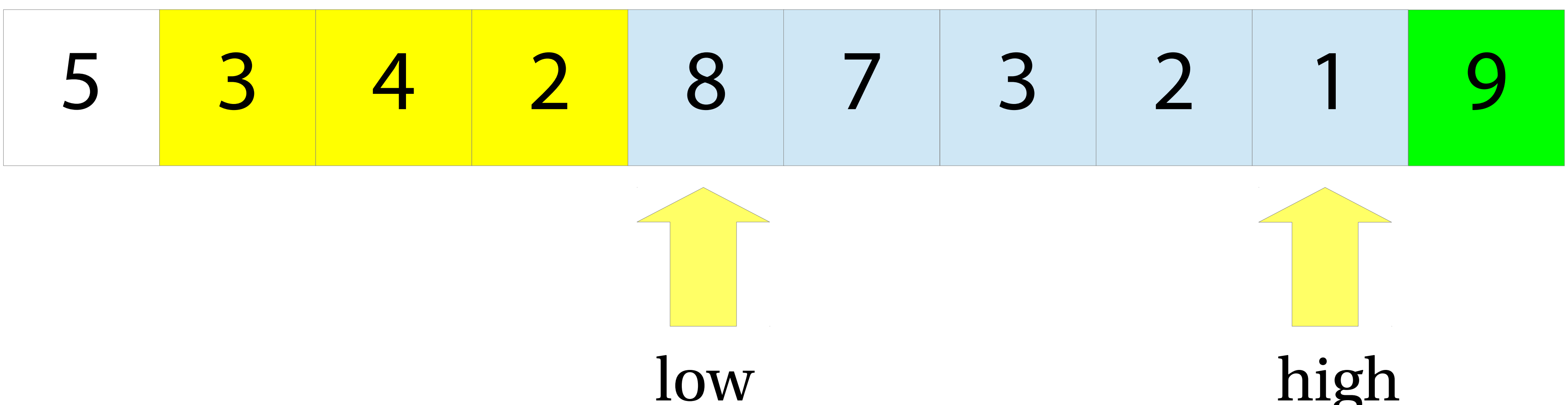
5. Advance low and high and repeat




```
low++; high--;
```

Partitioning algorithm

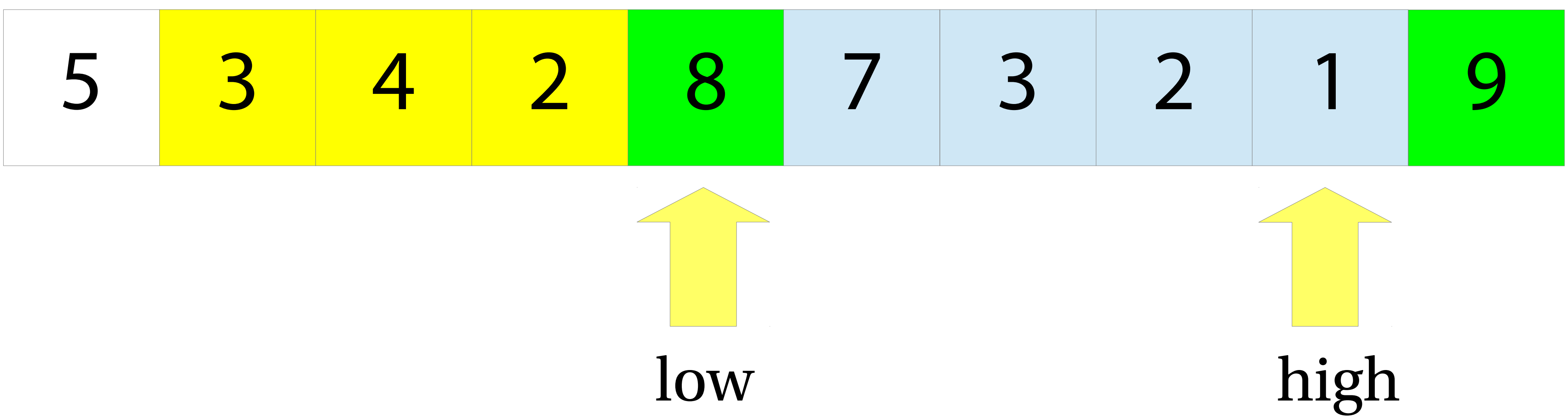
5. Advance low and high and repeat




```
while (a[low] < pivot) low++;
```

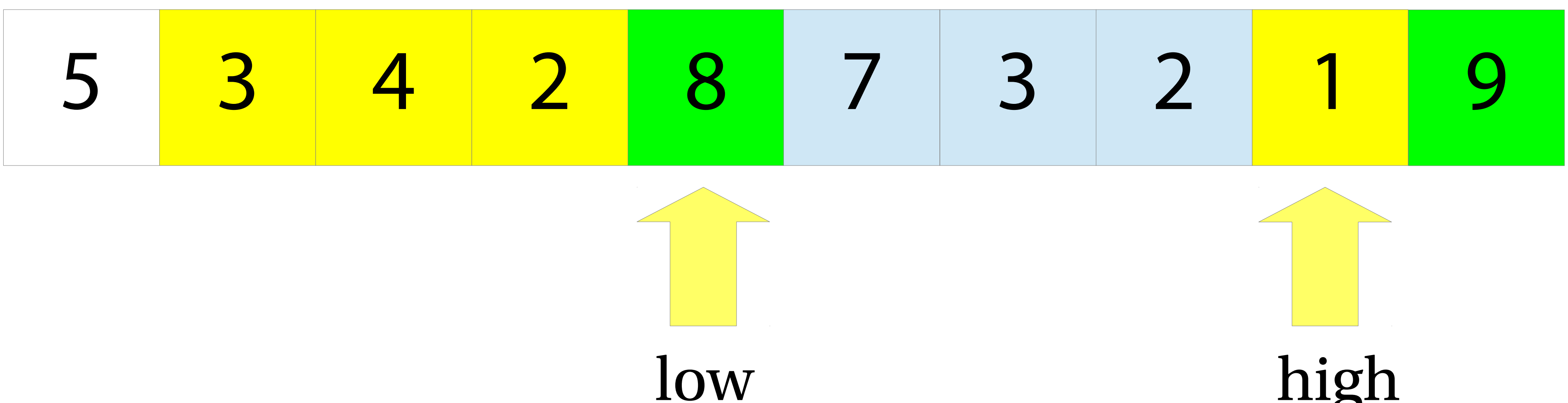
Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

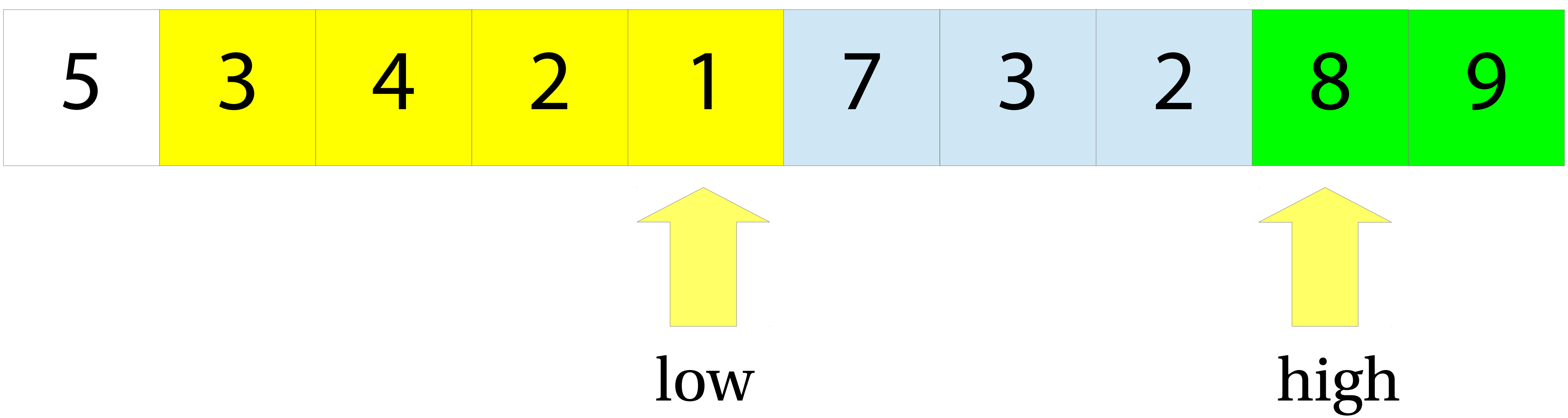
5. Advance low and high and repeat




```
while (a[high] < pivot) high++;
```

Partitioning algorithm

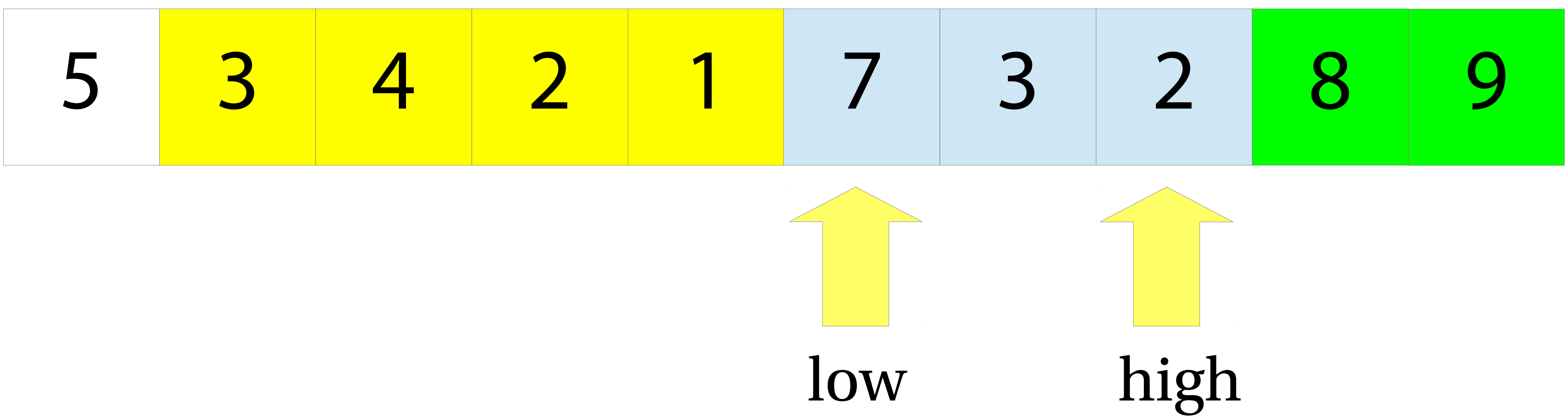
5. Advance low and high and repeat




```
swap(a[low], a[high]);
```

Partitioning algorithm

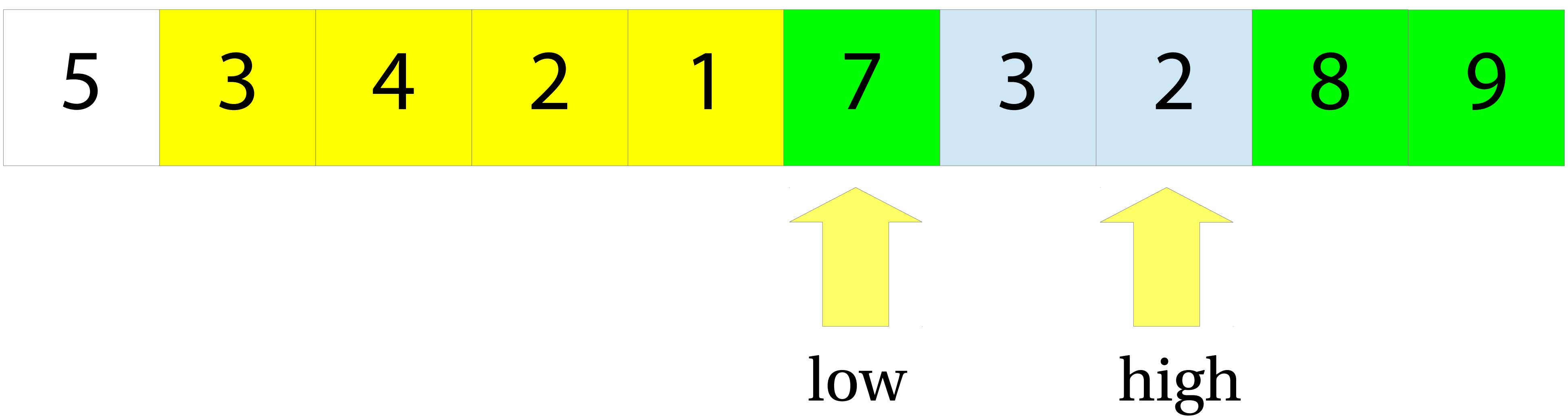
5. Advance low and high and repeat




```
low++; high--;
```

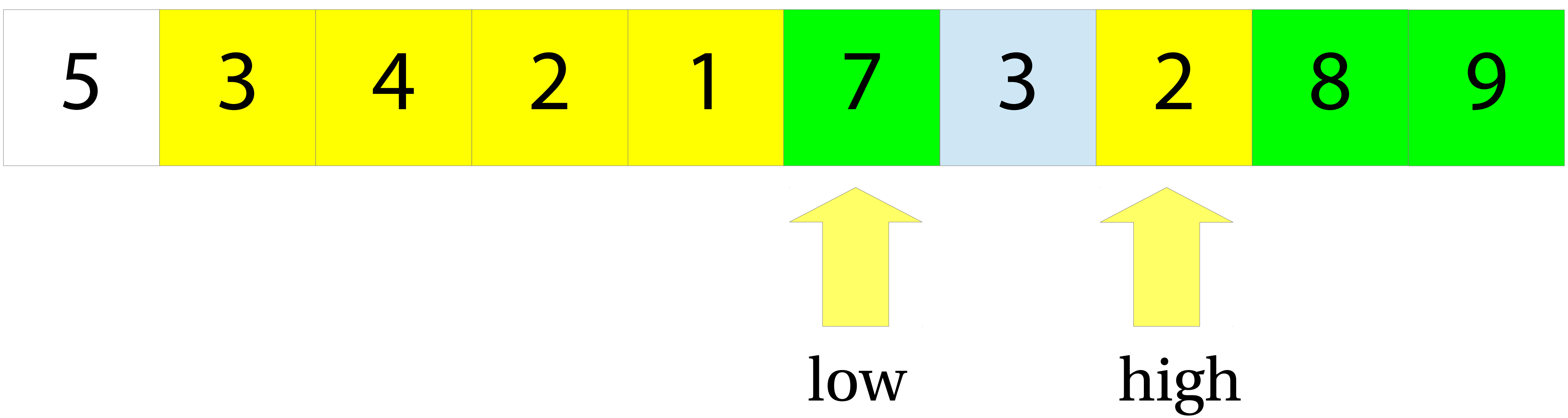
Partitioning algorithm

5. Advance low and high and repeat



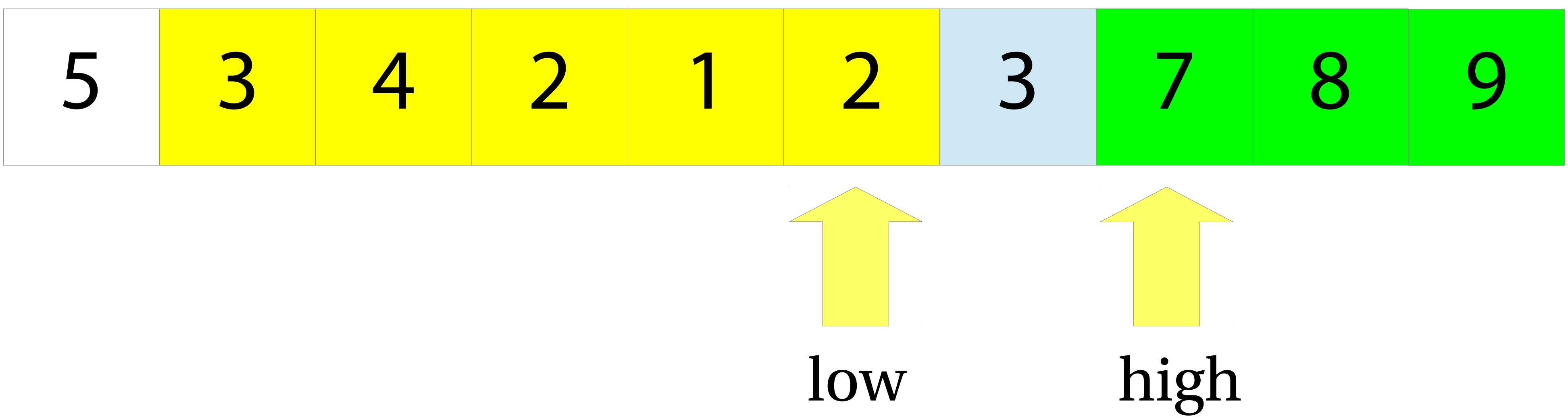
Partitioning algorithm

5. Advance low and high and repeat



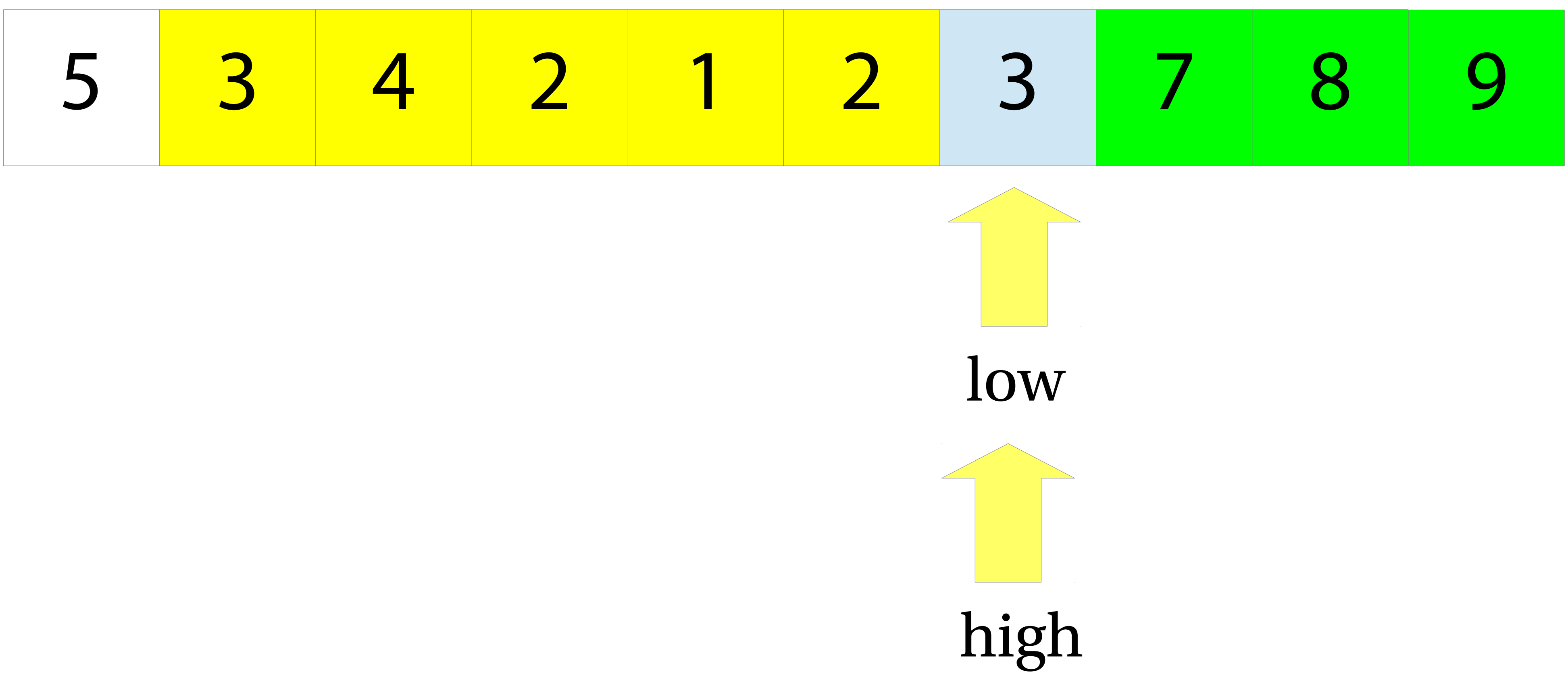
Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

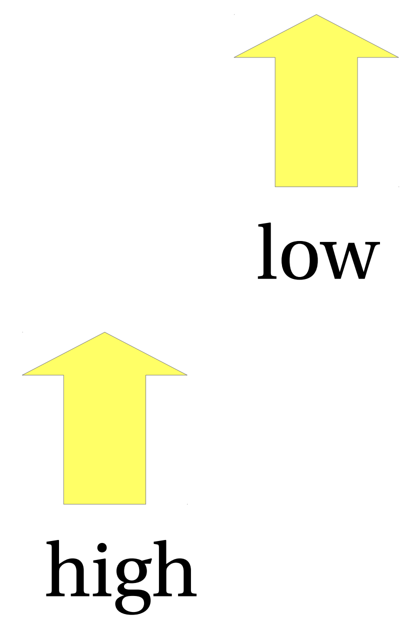
5. Advance low and high and repeat

Partitioning algorithm

6. When low and high have crossed, we are finished!

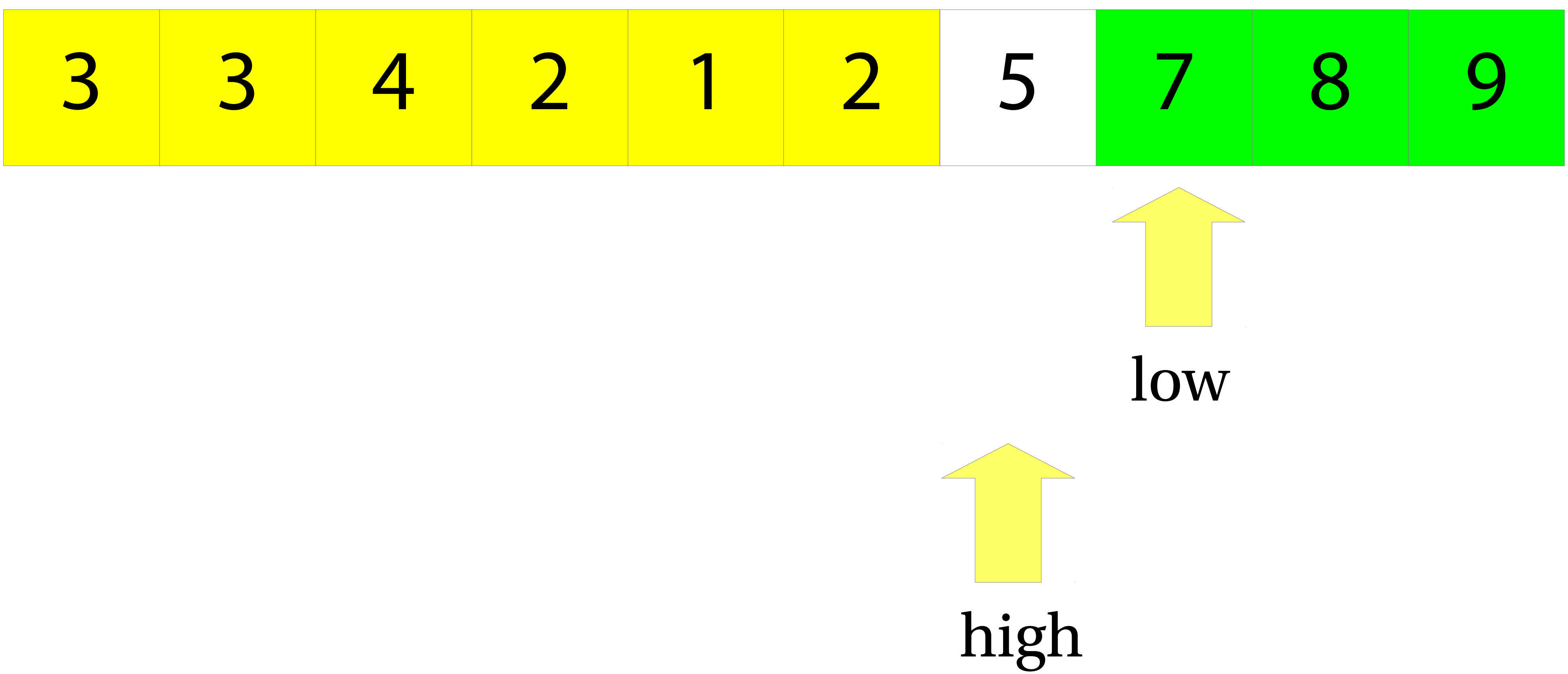
5	3	4	2	1	2	3	7	8	9

But the pivot is in the wrong place.



Partitioning algorithm

7. Last step: swap pivot with high



Details

1. What to do if we want to use a different element (not the first) for the pivot?

- Swap the pivot with the first element before starting partitioning!

Details

2. What happens if the array contains many duplicates?

- Notice that we only advance a[low] as long as a[low] < pivot
- If a[low] == pivot we stop, same for a[high]
- If the array contains just one element over and over again, low and high will advance at the same rate
- Hence we get equal-sized partitions

Pivot

Which pivot should we pick?

- First element: gives $O(n^2)$ behaviour for already-sorted lists
- Median-of-three: pick first, middle and last element of the array and pick the median of those three
- Pick pivot at random: gives O(n log n) *expected* (probabilistic) complexity

Quicksort

Typically the fastest sorting algorithm…
…but very sensitive to details!

- Must choose a good pivot to avoid $O(n^2)$ case
- Must take care with duplicates
- Switch to insertion sort for small arrays to get better constant factors

If you do all that right, you get an in-place sorting algorithm, with low constant factors and O(n log n) complexity

Mergesort vs quicksort

Quicksort:

- In-place
- $O(n \log n)$ but $O(n^2)$ if you are not careful
- Works on arrays only (random access)

Compared to mergesort:

- Not in-place
- $O(n \log n)$
- Only requires sequential access to the list – this makes it good in functional programming

Both the best in their fields!

- Quicksort best imperative algorithm
- Mergesort best functional algorithm

Sorting

Why is sorting important? Because sorted data is much easier to deal with!

- Searching – use binary instead of linear search
- Finding duplicates – takes linear instead of quadratic time
- etc.

Most sorting algorithms are based on *comparisons*

- Compare elements – is one bigger than the other? If not, do something about it!
- Advantage: they can work on all sorts of data
- Disadvantage: specialised algorithms for e.g. sorting lists of integers can be faster

Complexity of recursive functions

Calculating complexity

Let T(n) be the time mergesort takes on a list of size n

Mergesort does O(n) work to split the list in two, two recursive calls of size n/2 and O(n) work to merge the two halves together, so...

$$T(n) = O(n) + 2T(n/2)$$

Time to sort a list of size n

Linear amount of time spent in splitting + merging

Plus two recursive calls of size n/2

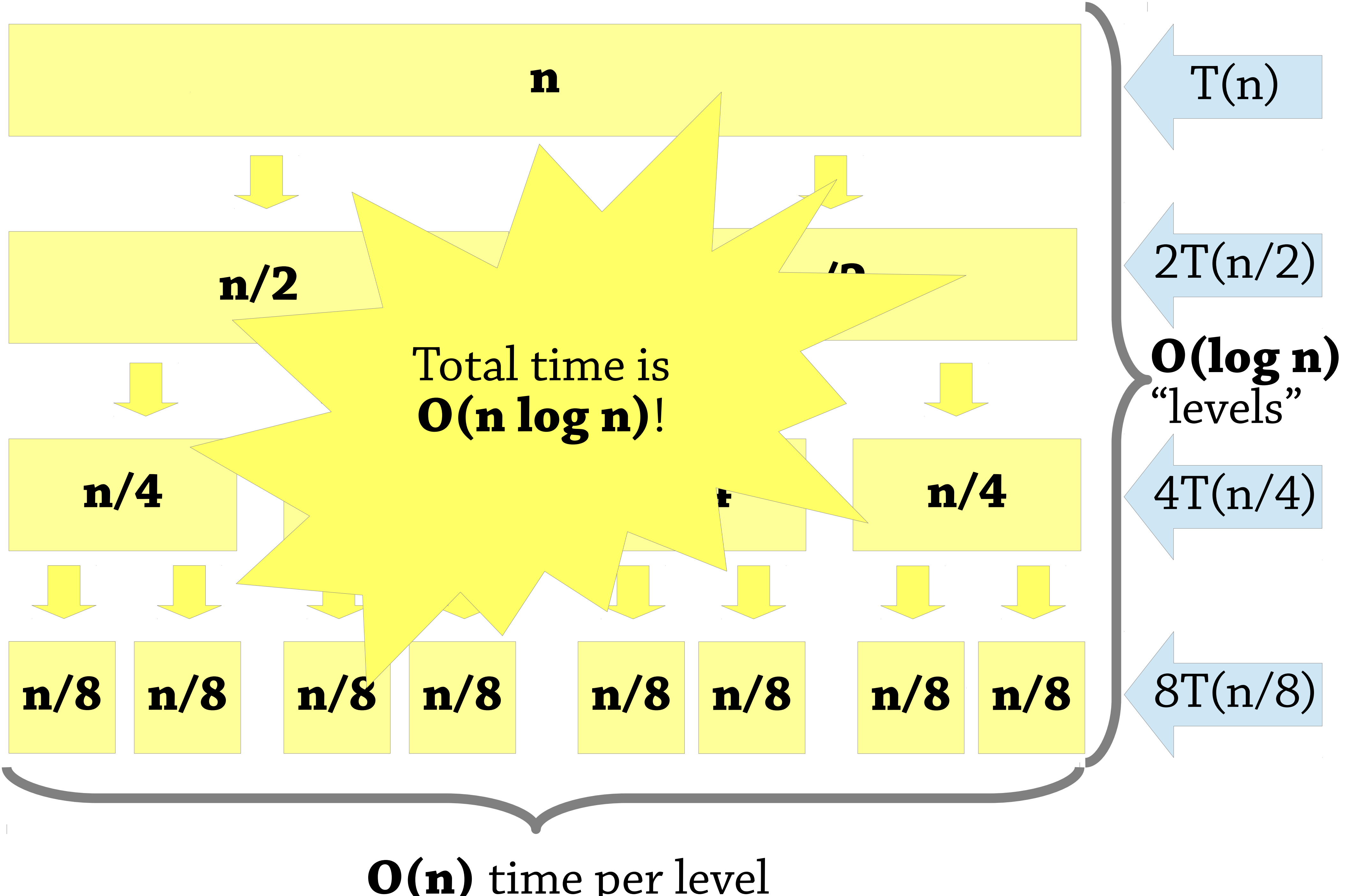
Calculating complexity

Procedure for calculating complexity of a recursive algorithm:

- Write down a *recurrence relation*
  e.g. T(n) = O(n) + 2T(n/2)
- *Solve* the recurrence relation to get a formula for T(n) (difficult!)

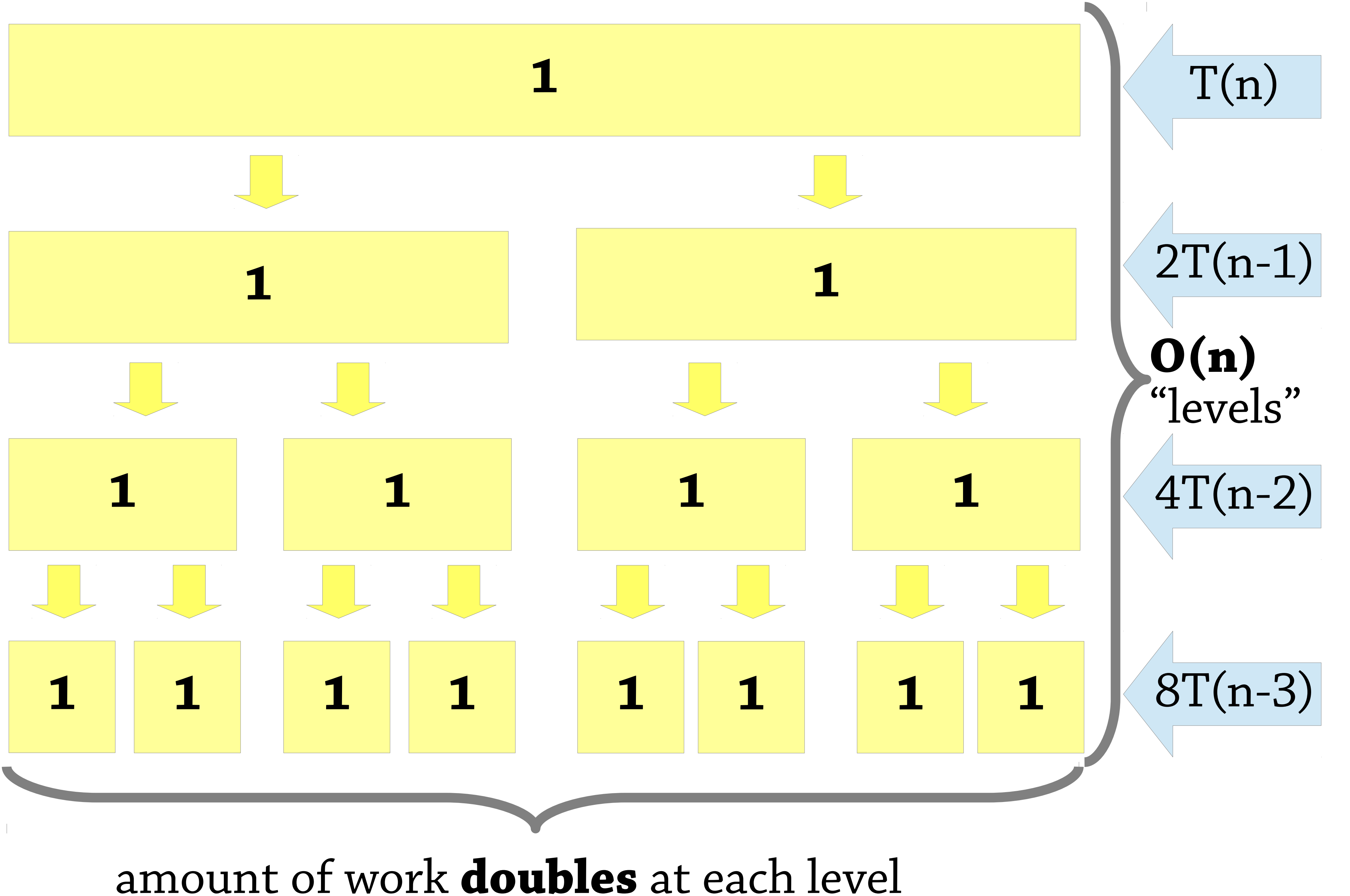
There isn't a general way of solving *any* recurrence relation – we'll just see a few families of them

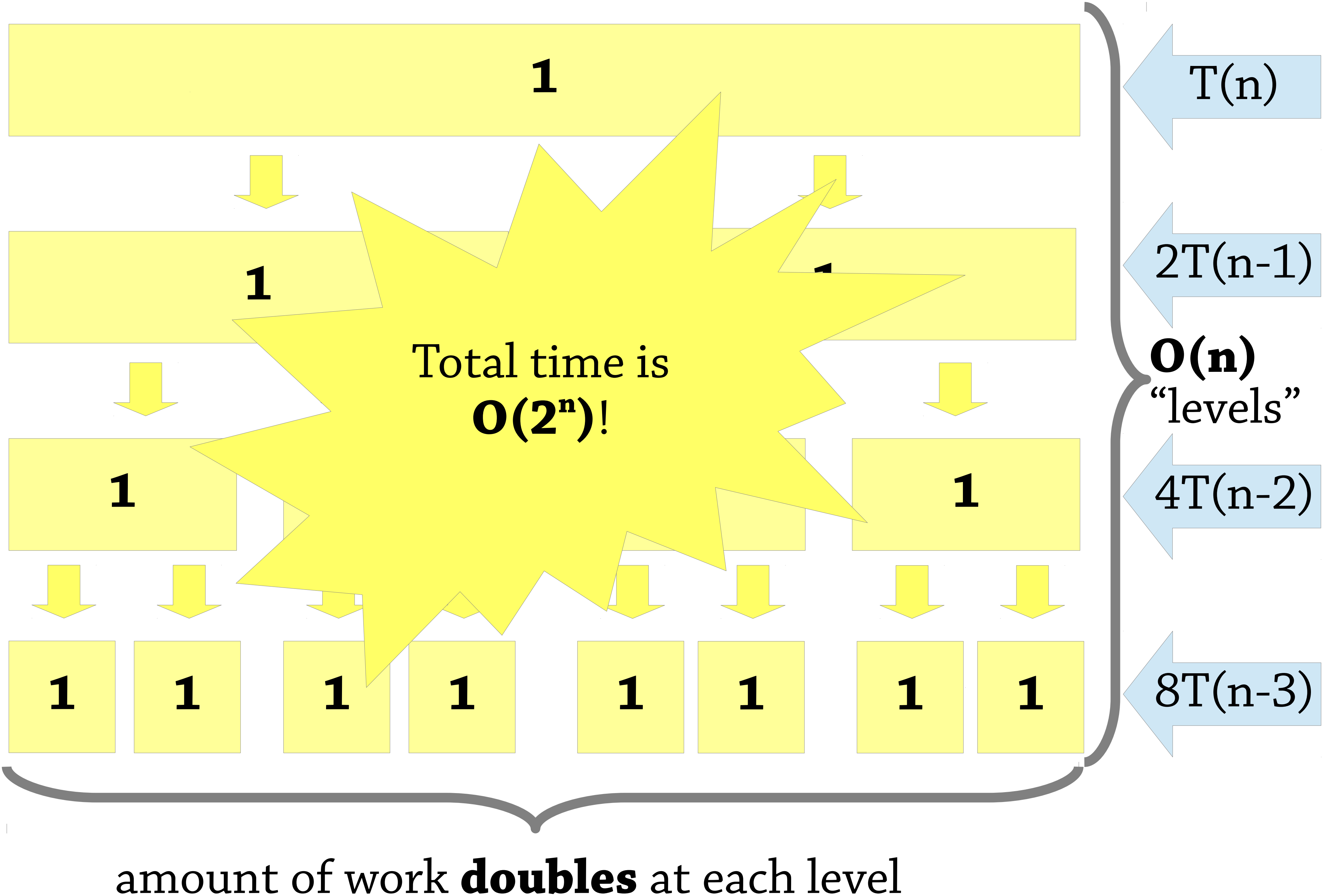
Approach 1:
draw a diagram


n
n/2
/2
n/4
n/4
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
T(n)
2T(n/2)
4T(n/4)
8T(n/8)
O(log n)
“levels”
Total time is
O(n log n)!
O(n) time per level

Another example:

$$T(n) = O(1) + 2T(n-1)$$


1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
T(n)
2T(n-1)
O(n)
“levels”
4T(n-2)
8T(n-3)
amount of work **doubles** at each level


1
1
1
1
1
1
1
1
1
1
1
1
Total time is
O(2ⁿ)!
T(n)
2T(n-1)
O(n)
"levels"
4T(n-2)
8T(n-3)
amount of work **doubles** at each level

This approach

Good for building an intuition

Maybe a bit error-prone

Approach 2: *expand out* the definition

Example: solving $T(n) = O(n) + 2T(n/2)$

Expanding out recurrence relations

$$T(n) = n + 2T(n/2)$$

Get rid of big-O before expanding out (n instead of O(n)) – the big O just gets in the way here

Expanding out recurrence relations

$T(n) = n + 2T(n/2)$

$= n + 2(n/2 + 2T(n/4))$

Expand out T(n/2)

$= n + n + 4T(n/4)$

$= n + n + n + 8T(n/8)$

$= \ldots$

$= n + n + n + \ldots + n + T(1)$ (log n times)

$= O(n \log n)$

(Note that T(1) is a constant so O(1))

If you prefer it a bit more formally...

$T(n) = n + 2T(n/2)$

$= 2n + 4T(n/4)$

$= 3n + 8T(n/8) = \ldots$

General form is $\mathbf{kn + 2^k T(n/2^k)}$

When $k = \log n$, this is $\mathbf{n \log n + nT(1)}$

which is $O(n \log n)$

Divide-and-conquer algorithms

$T(n) = O(n) + 2T(n/2)$: $T(n) = O(n \log n)$

- This is mergesort!

$T(n) = 2T(n-1)$: $T(n) = O(2^n)$

- Because 2^n recursive calls of depth n (exercise: show this)

Other cases: *master theorem* (Wikipedia)

- Kind of fiddly – best to just look it up if you need it

Another example: $T(n) = O(n) + T(n-1)$

$$T(n) = n + T(n-1)$$

$$= n + (n-1) + T(n-2)$$

$$= n + (n-1) + (n-2) + T(n-3)$$

$$= \ldots$$

$$= n + (n-1) + (n-2) + \ldots + 1 + T(0)$$

$$= n(n+1) / 2 + T(0)$$

$$= O(n^2)$$

Another example: $T(n) = O(1) + T(n-1)$

$T(n) = 1 + T(n-1)$

$= 2 + T(n-2)$

$= 3 + T(n-3)$

$= \ldots$

$= n + T(0)$

$= O(n)$

Another example: $T(n) = O(1) + T(n/2)$

$T(n) = 1 + T(n/2)$

$= 2 + T(n/4)$

$= 3 + T(n/8)$

$= \ldots$

$= \log n + T(1)$

$= O(\log n)$

Another example: $T(n) = O(n) + T(n/2)$

$T(n) = n + T(n/2)$:

$T(n) = n + T(n/2)$

$= n + n/2 + T(n/4)$

$= n + n/2 + n/4 + T(n/8)$

$= \ldots$

$= n + n/2 + n/4 + \ldots$

$< 2n$

$= O(n)$

Functions that recurse once

$T(n) = O(1) + T(n-1)$: $T(n) = O(n)$

$T(n) = O(n) + T(n-1)$: $T(n) = O(n^2)$

$T(n) = O(1) + T(n/2)$: $T(n) = O(\log n)$

$T(n) = O(n) + T(n/2)$: $T(n) = O(n)$

An *almost-rule-of-thumb*:

- Solution is *maximum recursion depth* times *amount of work in one call*

(except that this rule of thumb would give $O(n \log n)$ for the last case)

Complexity of recursive functions

Basic idea – recurrence relations

Easy enough to write down, hard to solve

- One technique: expand out the recurrence and see what happens
- Another rule of thumb: multiply work done per level with number of levels
- Drawing a diagram might help

Master theorem for divide and conquer

Luckily, in practice you come across the same few recurrence relations, so you just need to know how to solve those